Preprint
UCRL-JC-145580

A Quantitative Measure of Memory Reference Regularity

T. Mohan, B. R. de Supinski, S. A. McKee, F. Mueller, A. Yoo

This article was submitted to:
International Parallel and Distributed Processing Symposium, Fort Lauderdale, Florida, April 15-19, 2002

October 1, 2001

Approved for public release; further dissemination unlimited

A Quantitative Measure of Memory Reference Regularity

Tushar Mohan[1], Bronis R. de Supinski[2], Sally A. McKee[1], Frank Mueller[3] and Andy Yoo[2]

[1] School of Computing
University of Utah
Salt Lake City, UT 84112
{*tushar, sam*}*@cs.utah.edu*

[2] Lawrence Livermore National Laboratory
Center for Applied Scientific Computing
Livermore, CA 94551
{*bronis, ayoo*}*@llnl.gov*

[3] Department of Computer Science
North Carolina State University
Raleigh, NC 27695-7534
mueller@cs.ncsu.edu

Abstract

The memory performance of applications on existing architectures depends significantly on hardware features like prefetching and caching that exploit the locality *of the memory accesses. The principle of locality has guided the design of many key micro-architectural features, including cache hierarchies, TLBs, and branch predictors. Quantitative measures of spatial and temporal locality have been useful for predicting the performance of memory hierarchy components. Unfortunately, the concept of locality is constrained to capturing memory access patterns characterized by proximity, while sophisticated memory systems are capable of exploiting other predictable access patterns.*

Here, we define the concepts of spatial *and* temporal regularity*, and introduce a measure of spatial access regularity to quantify some of this predictability in access patterns. We present an efficient online algorithm to dynamically determine the spatial access regularity in an application's memory references, and demonstrate its use on a set of regular and irregular codes. We find that the use of our algorithm, with its associated overhead of trace generation, slows typical applications by a factor of 50-200, which is at least an order of magnitude better than traditional full trace generation approaches. Our approach can be applied to the characterization of program access patterns and in the implementation of sophisticated, software-assisted prefetching mechanisms, and its inherently parallel nature makes it well suited for use with multi-threaded programs.*

1 Introduction

Processor speeds are increasing much faster than memory speeds, and this disparity prevents many applications from making effective use of the tremendous computing power of modern microprocessors. Current access times of 50 cycles or more often cause memory performance to dominate application run time, and

This work was performed under the auspices of the U.S. Department of Energy by the University of California Lawrence Livermore National Laboratory under Contract W-7405-Eng-48, UCRL-JC-145580 and National Science Foundation award 0073532.

the processor/memory performance gap continues to grow. Streamed computations with strided access patterns are among those whose performance suffers most acutely. The vectors used in such applications lack temporal and often spatial locality, and thus have poor cache behavior. Nonetheless, these access patterns have the advantage of being predictable, and this predictability can be exploited to improve the efficiency of the memory subsystem.

Attempts to mitigate the memory bottleneck range from hiding or reducing latency to increasing effective memory bandwidth. Latency-hiding approaches like prefetching encompass a broad range of memory access techniques involving software, hardware, or both. The purely software approach relies on a compiler to generate instructions to preload data [26, 25], or an application writer to modify source code to achieve the desired behavior [5, 28, 21]. Hybrid approaches include hardware support for prefetch operations, exposing those mechanisms to software. For instance, they might augment the ISA with a prefetch instruction [12], redefine a load to a specific register (e.g., to register 0, as in the PA-RISC architectures [20]), or provide programmable prefetch engines [8] or programmable stream buffers [23]. Baer and Chen [2], Fu and Patel [14], and Sklenar [30] propose dynamic vector prefetch units that induce stream parameters at run time. The cache-based sequential hardware prefetching of Dahlgren *et al.* [11] eliminates the need for detecting strides dynamically. To minimize the number of unnecessary prefetches, the prefetch distance of these run-time techniques is generally limited to a few loop iterations (or a few cache lines). Hardware-only prefetching [2, 19, 11, 13, 30] has the advantage of being transparent, but because of its speculative nature, care must be taken to keep from lowering application performance by increasing contention in the caches and wasting bus bandwidth on useless prefetches.

Dynamic access optimization (DAO) techniques attempt to make better use of the memory system by changing the order and/or apparent location of memory references. For instance, the Impulse memory system can significantly improve the performance of applications with predictable access patterns but poor spatial or temporal locality [7]. This approach reduces observed memory latency by improving processor cache utilization and memory bus utilization, and it masks latency by prefetching data within the memory controller. Other work to improve effective memory bandwidth has examined memory scheduling mechanisms in the context of compiler-supplied or application-supplied information about access patterns, either in uniprocessors [22, 35, 23] or vector machines [10].

Quantitative measures of of a program's locality can help to predict application performance on architectures that employ some of these techniques, and, more importantly, can be leveraged to guide the profitable use of these mechanisms. Here we extend the principle of locality to the principle of *regularity*. Informally, *spatial regularity* is the tendency of programs to continue patterns of previous accesses, and *temporal regularity* is the tendency of programs to repeat previous sets of accesses. In the sections that follow, we define a simple metric for quantifying spatial regularity, and we present an algorithm for dynamically detecting the occurrence of streams. We then use this algorithm to compute our spatial regularity metric for a set of benchmarks ranging from micro-benchmarks to an application representative of important codes in use at Lawrence Livermore National Laboratory. Our approach can be applied to the characterization of program access patterns and in the implementation of sophisticated, software-assisted prefetching mechanisms, and its inherently parallel nature makes it well suited for use with multi-threaded programs.

2 The Principle of Regularity

The principle of locality is well known and recognized for its importance in determining application performance. Traditionally, this principle is considered to have two primary components, and a classic definition of these is found in Hennessy and Patterson's computer architecture text [16]:

- *Temporal locality* (locality in time) — If an item is referenced, it will tend to be referenced again soon.

- *Spatial locality* (locality in space) — If an item is referenced, nearby items will tend to be referenced soon.

Performance gains from traditional caching and prefetching are a testimony to the fact that reference locality exists in many applications, and locality measures [32, 15, 4, 34] can be helpful in predicting performance improvements from any such technique that attempts to exploit locality. Unfortunately, these concepts, and their measures, alone are insufficient to describe all the key application properties that intelligent memory systems can exploit. Consider the following code fragment:

```
double A[5000][5000];
int j, k;
. . .
for (j=0; j<5000; j++)
  for (k=0; k<5000; k++)              /* columnar stride */
    A[k][j] = A[k][j] * A[k][j];      /* square the element */
```

Assuming row-major layout, the columnar strides cause successive iterations of the inner loop to access array elements that are widely spaced in memory. On traditional prefetching systems, few (if any) performance gains can be expected through data prefetches when executing this fragment. The temporal locality in the reference sequence will yield some benefit from caching, though. For such a case, locality measures provide an accurate assessment of code speed-up (or lack thereof) as a result of caching and prefetching on traditional controllers.

Note that in spite of the lack of spatial locality, a very regular pattern of data references exists here. This pattern can be exploited by smarter memory subsystems that (possibly with the help of the compiler) detect the stream, prefetch successive stream elements (either within the controller or into cache), and perhaps gather them into dense cache blocks [2, 14, 30, 8, 23, 22, 35]. Locality metrics do not give an accurate depiction of application performance in the presence of such memory subsystems.

In order to describe some of the additional program characteristics that these advanced memory systems attempt to exploit, we extend the principle of locality to the principle of regularity. As with locality, regularity can be broken into two components:

- *Temporal regularity* (regularity in time) — If a set of items is referenced, it will tend to be referenced again soon.

- *Spatial regularity* (regularity in space) — If a set of items referenced establishes a predictable pattern, items that continue that pattern will tend to be referenced soon.

Temporal regularity simply extends temporal locality to sets of items, and it reduces to temporal locality when the set consists of a single item. Spatial regularity extends spatial locality to patterns of accesses; our intent is to capture the notion of streams of references in an issuing sequence such as strided array accesses. Spatial regularity reduces to spatial locality when the pattern is quickly established and the stride is small.

Most computer architecture features that exploit regularity depend on the program's exhibiting spatial regularity. However, compiler-based approaches for exploiting temporal regularity have been identified [9], and we expect that other, hardware-based techniques will emerge as the principle of regularity becomes better understood. Nonetheless, we focus on spatial regularity for the remainder of this paper, since it is more commonly exploited and is more tractable to measure.

3 A Spatial Regularity Metric

In order to measure spatial regularity, we must refine our concept of a *pattern* that a set of referenced items can exhibit. First, we define a *regular sequence* as a sequence of locations (or numbers, more generally) in which there exists a functional relationship between constituting members. From a theoretical standpoint, there are few restrictions on what the defining relationship may be. Practically, however, there is a limit on the ability to identify complex streams (and on the complexity of a hardware mechanism to exploit such streams). For our purposes, we will assume that only addresses in an arithmetic progression constitute a regular sequence, as in $x_{n+1} = x_n + c$, where c is a constant and x_i is the i^{th} reference in the regular sequence. We choose this definition because of the widespread occurrence of such sequences in common codes, as well as the acceptable complexity in identifying them. As an illustration, consider the following series of numbers (or access addresses): `0 4 8 12 16 20`. This constitutes a regular sequence with stride four, length six and starting element `0`. Hereafter, we will use the terms *stream* and *regular sequence* interchangeably. The following metric can be used to quantify the spatial regularity of an application:

$$R_{spatial} = \frac{\sum \mid s_i \mid}{N}$$

where $\mid s_i \mid$ is the length of the i^{th} sequence and N is the total number of references. If we do not allow a reference to be included in more than one sequence (i.e., we disallow multiple membership), the metric is a positive number not greater than unity. Higher metric values imply greater spatial regularity, and our example code segment above exhibits high regularity: each memory reference in the code segment belongs to a stream, and consequently the metric for the code is exactly one. In addition to the above metric, statistics such as the mean length of sequences or variance in sequence lengths can be used to obtain a more accurate picture of an application's regularity.

4 An Online Algorithm to Measure Spatial Regularity

One approach to determining spatial regularity performs static analysis of the code to recognize sections that access elements in a stream, as in loop nests that iterate through an array. Code can be inserted to record actual stream lengths at run time, as in the WM compiler [3], and to compute the regularity metric dynamically. Another approach instruments the code at compile time to record details of actual loads and stores at run time, inserting a call to a subroutine before every memory reference and passing the referenced address and the type of access (read or write) as parameters. The subroutine can choose to process the data online or to archive it for later processing. Large amounts of trace data make the option of archiving traces unacceptable for all but very small program runs.

Note that these two methods may compute different regularity values for the same program because of the different information available at compile time versus run time:

- Static approaches do not have the benefit of information such as de-referenced addresses of pointers, since these can only be available at runtime.

- Static approaches are generally constrained to detecting regularity within a basic block, rather than across multiple blocks.

- The practical limitations on algorithms that use trace data for stream detection render it difficult to detect streams for which successive elements do not fall within a fixed size *window* of subsequent references. Any dynamic, online algorithm for stream detection from traces must employ such a window, and we expect off-line algorithms (if they run in reasonable time) to have a similar limitation. Static approaches have no such restriction.

Our approach to measuring spatial regularity hinges on an efficient *online* algorithm — both in terms of space and time complexity — to detect streams by analyzing the trace data of a program's loads and stores. Our dynamic stream-detection algorithm assumes that streams are comprised solely of regular sequences.

In order for our algorithm to be useful in analyzing real programs, we must allow streams to be interleaved with each other and with non-stream references. For example, consider:

```
sum = 0;
for (i=0; i< MAX_ELEMS; i++)
  sum += A[i];
```

If we assume `sum` does not reside in a register, then the accesses to `A`, which form a stream, will be interleaved with references to `sum`. We would like our algorithm to detect the stream corresponding to `A` even in the presence of the alternating accesses to `sum`. In other words, in the access sequence `a b c d p z e f`, we would like to identify the regular sequence `a b c d e f`, despite the noise of `p z`. Constraints on space require us to periodically discard older trace data to make room for newer references. This *aging* prevents the detection of streams interspersed with a large number of intervening accesses. Taking advantage of the regularity in such largely spaced streams (i.e., with little locality) is likely to be much more difficult, and so we find this restriction to be reasonable for our purposes.

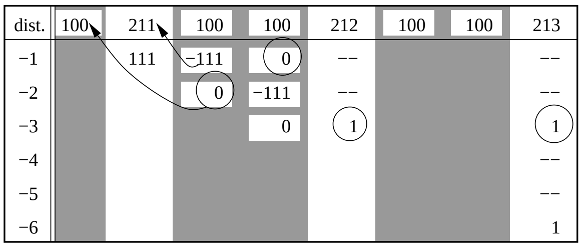
```
WHILE new reference exists DO
  Increment column; /* move window */
  /* Add reference to pool */
  pool[0][column] := new reference;
  IF reference extends stream IN stream table  THEN
    Update length of stream in stream table;
    Mark column in pool (shaded in example);
  ELSE
    /* Compute and store differences in pool */
    FOR i := 1 TO window size DO
      pool[i][column] := pool[0][column] - pool[0][column-
i];
    END FOR;
    found := FALSE;
    /* Search for streams of minimum length 3 */
    FOR i := 1 TO window size DO
      FOR k := 1 TO window size DO
        IF pool[i][column] == pool[k][column-i] THEN
          found = TRUE;
        END IF;
      UNTIL found;
    UNTIL found;
    IF found THEN
       Enter stream in stream table;
       Mark corresponding columns in pool (shaded in example);
    END IF;
  END IF;
END WHILE;
```

Figure 1. Online Algorithm to Detect Streams

The algorithm requires maintaining two data structures:

- **Stream Table**: This contains a compact description of streams that have already been detected. The table is stored as a chained hash, with the *expected successor to the stream* serving as the hash key. Each node in the table is a triple consisting of the `start address`, `length`, and `stride` of the stream. If aging of stream elements is desired, an additional value representing the age of the last stream element can be added to the record. An additional field may also be needed in the record to implement chaining in the hash table. Since these are specific to the implementation, and not germane to the working of the algorithm, we assume that the record consists of only the three elements described above.

- **Pool**: This contains the unaged references that have not yet been detected to be part of a stream. These references form the window of addresses scanned for potential streams. As new addresses are referenced, the window of active addresses expands to fill the storage structure of the pool. Once filled, the addition of each new reference causes an earlier reference to be discarded after a search for streams. This process of addition and removal from the pool can be viewed as the cyclical rotation of the active window through the pool's storage structure. In determining the existence of streams, elements of which may be separated by arbitrary numbers of intervening accesses, it is necessary to compute *differences* between elements of the pool. To reduce the computational complexity in repeatedly determining the differences between existing elements as new elements are added, we store a set of differences with each reference. Given this pool structure, detecting streams becomes a matter of finding a sequence of elements such that the differences between successive elements match. In our implementation, the pool is organized as a statically allocated, two-dimensional array along with two indices, the *start* and the *end* of the active addresses. The indices advance via modulo arithmetic through the pool.



dist.	100	211	100	100	212	100	100	213
−1		111	−111	0	−−			−−
−2			0	−111	−−			−−
−3				0	1			1
−4								−−
−5								−−
−6								1

Figure 2. Snapshot of the Reservation Pool

Multi-threaded programs can easily employ our stream detection algorithm. A unique thread running this algorithm (and maintaining its own stream table and pool) can be dedicated to receiving the accesses of a corresponding thread in the instrumented application. At the conclusion of stream detection of all threads, the data can be collected to compute the regularity metric and other statistics on a per-thread basis or for the application as a whole. By maintaining separate data structures in each thread, the detection of streams across application threads is not possible, but we believe this is desirable. By maintaining separate stream tables, implementations could also distinguish between loads and stores, since these are often handled differently in modern architectures. An ancillary benefit of this stream detection process is that it can be leveraged to quantify temporal locality, which corresponds to streams with zero stride, and temporal regularity, which depends on the extent of stream repetition. The algorithm, omitting the details of aging and the distinguishing between access types, is presented in Figure 1. We illustrate the application of our algorithm on the following highly regular sequence of accesses, where we have omitted all intervening accesses for simplicity:

`100, 211, 100; 100, 212, 100; 100, 213, 100; 100, 214, 100; ...`

The pool can be viewed as a table as shown in Figure 2, which depicts a snapshot of the pool after encountering the first eight references. The header row shows the referenced locations. Each column contains the difference between the value in the current column header and the value in a preceeding column (see "compute and store differences" in Figure 1). The particular element used for calculating the difference depends on the row for which the difference is to be computed. The first row (below the header) consists of the difference between an element and its immediate predecessor (distance -1), exemplified by the upper arrow. The second row consists of differences between an element and its penultimate predecessor (distance -2).

To capture streams within a window size w, we need only compute the differences above the diagonal of the pool table. Elements determined to be part of a stride are removed from the table. In the example above, on seeing the third `100` (assuming a minimum length of three), we identify a stream by observing the two corresponding differences of `0` (circled) in a transitive relationship. Consequently, we insert a stream of `<100,3,0>` in the stream table. These elements are shown shaded in the pool to illustrate their absence from the subsequent difference computations. Similarly, the later `100`s are immediately observed to belong to the stream, and the stream fields are modified to `<100,5,0>` on receiving the fifth `100`. To preserve the notion of window size, we still keep a slot for an element detected to be a part of a stream, but we omit differences for that element. On seeing `213`, a new stream is identified by observing an identical difference of `1` (circled) for the transitive relation between 211, 212 and 213. At this point, `<211,3,1>` is inserted in the stream table, and the slots for these elements can be marked to indicate their non-participation in further stream detection.

Although not explicitly shown in the outline of the algorithm, we implicitly assume that sequences must have a minimum length to qualify as streams. The value of metrics computed on streams is dependent on this parameter. The worst case complexity of the algorithm is $O(N * w^2)$, where N is the number of total reference and w is the window size. This can be reduced further if differences with aged elements are not computed. Our algorithm intentionally prevents membership of an element in multiple streams. We enforce this by removing elements from the pool on detecting their membership in a stream.

5 Implementation and Results

We have implemented our algorithm for both Linux-x86 and Solaris-x86 platforms. The Portland Group, Inc., C and Fortran compilers [29] were used to instrument our benchmarks such that all loads and stores to memory are preceeded by a call to a special subroutine to which the referenced address and access type (load/store) are passed as arguments. The subroutine forwards this information to a separate process that implements our stream detection algorithm. Unix FIFOs were used for inter-process communication, although other mechanisms such as shared memory could also be used. Employing a separate process for stream detection decouples the instrumented application from the detection process as much as possible. Our implementation of the stream detection algorithm is itself multi-threaded, and handles multi-threaded applications with ease: we merely devote a separate thread in the detection process to each application thread, and each detection thread maintains separate data structures. Stream aging is optional, and we do not currently differentiate between loads and stores in determining stream membership. This simplified the analysis of initial results, but distinguishing among different access types can be easily implemented by maintaining separate data structures.

We ran our stream detection algorithm on a variety of regular and irregular codes, on both, an AMD-K6 machine running Linux, and the SRC-6 [31], which is an SMP x86 system running Solaris. For our initial experiments, we compiled the benchmarks with global and loop optimizations disabled. Application run-times increased between one and two orders of magnitude, depending on the extent of memory activity versus computation. We found that to reduce noise in the detected streams, the minimum length qualifying a series of accesses to be a stream should be set higher than three. Setting the pool width too low can also significantly affect which streams are detected and at which lengths. For instance, when the pool width is close to the minimum stream length, many streams obviously remain undetected. For most of our tests, we found a minimum stream length of five and a pool width of 60 to work well. We also found that aging of streams significantly affects the streams detected and their observed lengths. For all the initial results presented here, stream aging was disabled. Nonetheless, the definition of the regularity metric makes it relatively insensitive to changes in detected streams lengths, since the total number of regular references rarely changes dramatically — only the stream membership changes.

Table 1 lists the results obtained by applying the algorithm to single-threaded versions of representative codes. Two of our tests — *daxpy* and *matmult* — were parameterized: *daxpy* was executed with vector sizes of 1024, and *matmult* performed multiplication on $100X100$ matrices. For the obviously regular *daxpy* and *matrix multiplication*, a high value of regularity (> 0.95) was observed, with the expected streams being detected. The mean stream lengths for these two are high — another indicator of regularity. The large standard deviation in stream lengths for *matmult* results from a disparity in the stream lengths: loop variables contributed extremely long zero-stride streams, while the array accesses contributed far shorter streams. This is also reflected in the lower mean and standard deviation, when we exclude zero-stride streams. Irregular codes like UMT98[1] — which has a substantial use of indirection vectors and is a fairly accurate representation of some of the scientific codes used at LLNL [2] in terms of its use of memory — exhibited a far lower regularity metric. The lower mean stream lengths also corroborates the lack of regularity. We ran our algorithm on multi threaded versions of *daxpy*, *UMT98* and *matmult*, with four threads. Since the parallelization of *daxpy* was simply a work-sharing OpenMP directive which divided the loop equally among four threads, identical results for all threads were obtained for it. Table 2 lists the results for these runs.

Program	**Regularity**		**Mean Stream Length**		**Std. Dev. Stream Length**		**Slow-down**
	inc. 0-stride	ex. 0-stride	inc. 0-stride	ex. 0-stride	inc. 0-stride	ex. 0-stride	
umt98	0.65	0.35	64	19	682	272	73
matmult	1.00	1.00	600	198	1006	69	115
daxpy	1.00	1.00	10240	7680	58	51	37
bc	0.97	0.42	311	17	221	4	82
ks	0.48	0.34	17	9	264	2	210
anagram	0.94	0.67	178	22	484	42	47

Table 1. Regularity for single-threaded programs

Program	**Thread Num.**	**Regularity**		**Mean Stream Length**		**Std. Dev. Stream Length**	
		inc. 0-stride	ex. 0-stride	inc. 0-stride	ex. 0-stride	inc. 0-stride	ex. 0-stride
umt98	0	0.71	0.41	74	22	540.2	378.6
	1	0.59	0.29	45	13	352.9	57.2
	2	0.59	0.29	45	13	353.3	57.1
	3	0.59	0.29	45	13	353.1	57.2
matmult	0	1.00	1.00	399	132	475.9	11.5
	1	1.00	0.99	315	103	480.4	9.9
	2	1.00	0.99	320	105	507.0	9.9
	3	1.00	1.00	352	116	512.6	10.4
daxpy	0	1.00	1.00	2926	1920	30.2	25.2
	1	1.00	1.00	2926	1920	30.2	25.2
	2	1.00	1.00	2926	1920	30.2	25.2
	3	1.00	1.00	2926	1920	30.2	25.2

Table 2. Stream detection in multi threaded programs – 4 threads

[1]Unstructured Mesh Transport code
[2]Lawrence Livermore National Laboratory

6 Related Work

Traditional measures of locality include aggregate statistics such as cache hit rates, but these provide little insight as to why a program's access patterns generate the resulting cache behavior. Other measures break down summary performance data spatially or according to memory bandwidth requirements. For instance, Tyson *et al.* perform a detailed characterization of cache behavior for individual load instructions [33], and Abraham *et al.* study the predictability of memory referencing latencies for individual instructions [1]. Both studies confirm that a small number of loads account for a majority of data cache misses, and Thiebaut's application of fractal models to memory behavior illustrates that cache misses are bursty in nature [32]. Johnson *et al.* measure spatial reuse fractions for cache lines, finding that fetching data with a uniform, large cache block size usually wastes cache space and bus bandwidth [18]. Huang and Shen measure the bandwidth requirements of programs as a function of available local memory [17]. Burger *et al.* calculate traffic ratios, traffic inefficiencies, and effective pin bandwidths for different levels of the memory hierarchy [6]. McKinley and Temam take a step towards more detailed analysis by quantifying the locality characteristics of numerical loop nests [24].

Newer approaches to characterizing program locality make it easier to represent and discuss caching locality and behavior in concrete terms. Brehob and Enbody propose a mathematical model of locality that uses the distance between references in a trace to calculate a measure of temporal locality, and a correspondence to cache lines to represent spatial locality [4]. Grimsrud *et al.* introduce a method of quantifying the locality in a trace and visually representing it as a 3D surface [15]. They explore some of the properties of this formulation, showing the correlation between graphical features and specific reference patterns, and demonstrating the utility of their locality measure through two applications as a visualization tool: characterizing and summarizing workload locality, and evaluating benchmark effectiveness for exercising memory hierarchies. In contrast to these largely *ad hoc* approaches, Weikle develops an analysis methodology to provide theoretical foundations to support memory hierarchy design, and builds some of the mathematical and software tools to support this methodology [34]. Her framework yields greater insight into why memory hierarchies behave as they do for particular reference patterns, but the tools to fully automate the approach are not yet in place.

Chilimbi attempts to address the processor-memory performance gap by introducing the abstraction of *exploitable locality* — a combination of locality and regularity [9]. His definitions of regularity, both spatial and temporal, are sufficiently different from ours to merit special attention. His definition of regularity corresponds to our notion of temporal regularity, while his definition of temporal regularity has no counterpart in our work. Furthermore, his usage of "spatial regularity" differs significantly; we use it denote a property of programs, whereas he uses it to quantify a particular property of streams. Chilimbi focuses on non-scientific codes with references to scalar variables while we address non-scalar references with access patterns of varying regularity. While the specific development of regularity differs between his work and ours, the underlying theme is the same: the memory performance of applications can be improved by exploiting the presence of patterns in the memory references of an application.

7 Conclusions and Future Work

In an effort to better characterize application memory performance on modern architectures, we have defined a regularity metric to quantify the predictability of a program's reference patterns. To that end, we have extended the concept of program locality by formally defining the twin concepts of spatial and temporal regularity, and we have developed a metric to quantify the former. Calculating the metric for a particular program requires the detection of regular address sequences, or streams. In this paper, we restrict streams to be comprised of evenly spaced references, but more complex formulations are possible [27], and investigating them is part of future work. We have outlined an efficient, parallelizable algorithm to detect such streams dynamically, and one of the advantages of this online approach is the absence of a need for storing extremely large traces. The results for our implementation are promising, with increases in run-times between and 50 and 200. As expected, our metric reflects the high predictability of the accesses in the daxpy and matmult kernel, and indicates the lack of predictability in pointer benchmarks and irregular scientific computations. Our current work investigates the use of dynamic instrumentation [27] to selectively apply our approach to running programs, and the design of efficient memory system mechanisms to exploit the program regularity revealed by our metric.

References

[1] S. Abraham, R. Sugumar, D. Windheiser, B. Rau, and R. Gupta. Predictability of load/store instruction latencies. In *Proceedings of IEEE/ACM 26th International Symposium on Microarchitecture*, pages 139–152, Dec. 1993.

[2] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of Supercomputing'91*, pages 176–186, Nov. 1991.

[3] M. Benitez and J. Davidson. Code generation for streaming: An Access/Execute mechanism. In *Proceedings of the 4th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 132–141, Apr. 1991.

[4] M. Brehob and R. Enbody. A mathematical model of locality and caching. Technical Report TR-MSU-CPS-96-42, Michigan State University, Nov. 1996.

[5] J. Brooks. Single PE optimization techniques for the cray T3D system. In *Proceedings of the 1st European T3D Workshop*, Sept. 1995.

[6] D. Burger, J. Goodman, and A. Kagi. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 78–89, May 1996.

[7] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a smarter memory controller. In *Proceedings of the Fifth Annual Symposium on High Performance Computer Architecture*, pages 70–79, Jan. 1999.

[8] T.-F. Chen. Effective programmable prefetch engine for on-chip caches. In *Proceedings of IEEE/ACM 28th International Symposium on Microarchitecture*, pages 237–242, Nov. 1995.

[9] T. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 191–202, June 2001.

[10] J. Corbal, R. Espasa, and M. Valero. Command vector memory systems: High performance at low cost. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, pages 68–77, Oct. 1998.

[11] F. Dahlgren, M. Dubois, and P. Stenstrom. Sequential hardware prefetching in shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 6(7):733–746, July 1995.

[12] J. Edmondson, et al.˙ Internal organization of the alpha 21164, a 300-mhz 64-bit quad-issue cmos risc microprocessor. *Digital Technical Journal*, 7(1):119–135, 1995.

[13] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic. Memory-system design considerations for dynamically-scheduled processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 133–143, June 1997.

[14] J. Fu and J. Patel. Data prefetching in multiprocessor vector cache memories. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 54–65, Toronto, Canada, May 1991.

[15] K. Grimsrud, J. Archibald, R. Frost, and B. Nelson. Locality as a visualization tool. *IEEE Transactions on Computers*, 45(11):1319–1326, 1996.

[16] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1996.

[17] A. Huang and J. Shen. The intrinsic bandwidth requirements of ordinary programs. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 105–114, Oct. 1996.

[18] T. Johnson, M. Merten, and W. Hwu. Run-time spatial locality detection and optimization. In *Proceedings of IEEE/ACM 30th International Symposium on Microarchitecture*, pages 57–64, Dec. 1997.

[19] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture,* pages 364–373, May 1990.

[20] G. Kane. *PA-RISC 2.0 Architecture*, 1996.

[21] K. Lee. The NAS860 library user's manual. Technical Report NAS Technical Report RND-93-003, NASA Ames Research Center, Mar. 1993.

[22] B. Mathew, S. McKee, J. Carter, and A. Davis. Design of a parallel vector access unit for sdram memories. In *Proceedings of the Sixth Annual Symposium on High Performance Computer Architecture*, pages 39–48, Jan. 2000.

[23] S. McKee, W. Wulf, J. Aylor, R. Klenke, M. Salinas, S. Hong, and D. Weikle. Dynamic access ordering for streamed computations. *IEEE Transactions on Computers*, 49(11):1255–1271, Nov. 2000.

[24] K. McKinley and O. Temam. A quantitative analysis of loop nest locality. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems,* pages 94–104, Oct. 1996.

[25] L. Meadows, S. Nakamoto, and V. Schuster. A vectorizing software pipelining compiler for LIW and superscalar architectures. In *RISC'92*, pages 331–343, 1992.

[26] T. Mowry, M. S. Lam, , and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the 5th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, Oct. 1992.

[27] F. Mueller, T. Mohan, B. de Supinski, S. McKee, and A. Yoo. Partial data traces: Efficient generation and representation. In *Proceedings of the Workshop on Binary Translation*, Sept. 2001. held in conjunction with PACT'01.

[28] S. Palacharla and R. Kessler. Code restructuring to exploit page mode and read-ahead features of the cray T3D. Technical Report Internal Report, Cray Research, Feb. 1995.

[29] Portland Group, Inc. `http://www.pgroup.com/`, 2001.

[30] I. Sklenar. Prefetch unit for vector operation on scalar computers. *Computer Architecture News*, 20(4):31–37, Sept. 1992.

[31] SRC Computers, Inc. `http://www.srccomp.com/`, 2001.

[32] D. Thiebaut. On the fractal dimension of computer programs and its application to the prediction of cache miss ratio. *IEEE Transactions on Computers*, 38(7):1012–1026, July 1989.

[33] G. Tyson, M. Farrens, J. Matthews, and A. Pleszkun. A modified approach to data cache management. In *Proceedings of IEEE/ACM 28th International Symposium on Microarchitecture*, pages 93–103, Nov. 1995.

[34] D. Weikle. *Caches As Filters: A Framework for the Analysis of Caching Systems*. PhD thesis, University of Virginia, May 2001.

[35] C. Zhang and S. McKee. Hardware-only stream prefetching and dynamic access ordering. In *Proceedings of the 2000 International Conference on Supercomputing*, pages 167–175, May 2000.